Anti-Pattern Matching

Claude Kirchner, Radu Kopetz, Pierre-Etienne Moreau

Anti-Pattern Matching

Claude Kirchner, Radu Kopetz, Pierre-Etienne Moreau

INRIA & LORIA*, Nancy, France
{Claude.Kirchner, Radu.Kopetz, Pierre-Etienne.Moreau}@loria.fr

Abstract. It is quite appealing to base the description of pattern-based searches on positive as well as negative conditions. We would like for example to specify that we search for white cars that are not station wagons.
To this end, we define the notion of anti-patterns and their semantics along with some of their properties. We then extend the classical notion of matching between patterns and ground terms to matching between anti-patterns and ground terms. We provide a rule-based algorithm that finds the solutions to such problems and prove its correctness and completeness. Anti-pattern matching is by nature different from disunification and quite interestingly the anti-pattern matching problem is unitary. Therefore the concept is appropriate to ground a powerful extension to pattern-based programming languages and we show how this is used to extend the expressiveness and usability of the Tom language.

1 Introduction

Pattern matching is a widely spread concept both in the computer science community and in everyday life. Whenever we search for something, we build a structured object, a pattern, that specifies the features we are interested in. But we are often in the case where we want to exclude certain characteristics: typically we would like to specify that we search for white cars that are not station wagons.

We call *anti-patterns* the patterns that may contain complement symbols, denoted by $\neg$. For example, the web search engine from Google has an option where we can specify what specific words we do *not* want the result pages to contain. But it is not possible to express a search that has nested negations. What are the nested negations used for? Consider the following situation: using a search engine for cars, we want to search for a car that is not white; but in the case the car is ecological, we do not care about the color. This kind of search can be expressed in the following manner: $\neg car(white, \_) \vee car(\_, ecological)$ which could be equivalently expressed by the anti-pattern $\neg car(white, \neg ecological)$.

Another of our motivations comes from the popular "Business rules" management systems (BRMS for short) that provide a restricted anti-pattern capability. For example, although it is possible to use nested negations in Ilog JRules, one

* UMR 7503 CNRS-INPL-INRIA-Nancy2-UHP

of the most representative business rule language on the market[1], they are not handled in full generality. A BRMS consists mainly of three components: a set of facts representing the current state of the system called Working Memory (WM), a set of *IF-THEN* rules that test and alter the WM, and a rule interpreter that applies the rules on the WM. A BRMS uses pattern matching to find out if an object is in the WM or not. If we put in the working memory the following fact: *car(white, ecological)*, and we insert the following rules:

1. if *there is no car that has the color white and the type ecological* then $action_1$,
2. if *there is no car that has the color white and the type not ecological* then $action_2$,
3. if *there is no car that has the color white and the type not diesel* then $action_3$.

none of the actions are fired. When we look at the three rules, we can see that basically the rule engine ignores the second negation. We consider that for the second rule, the action should have been fired.

A further issue that is not addressed in current pattern matching based languages, is the problem of non-linearity inside a negative context. We are not aware of the existence of a language where we can express in a single pattern the following search: look for a car that does not have both interior and exterior color the same. This should give all the cars with different interior-exterior colors.

In this rich context, our first contribution is to define in the next section the concept of anti-pattern and its semantics. Indeed as a term t represents the set of all its ground instances, the anti-pattern $\neg t$ represents the complement of the representation of t in the set of ground terms and this definition is extended recursively. Of course, many frameworks and results have already contributed to the use of negation in logic based languages. Having in particular in mind negation by failure in Prolog [8], the explicit use of counter-examples [16], disunification [11], feature constraints [3], inclusion constraints [20] and negation in iRho [17], we will motivate and explain the usefulness of anti-patterns.

Our second contribution concerns the definition of the notion of matching anti-patterns against terms in Section 3. In Section 4 we present a rule based algorithm for transforming anti-pattern matching problems into classical equational ones. The latter ones can be further solved using a subset of the disunification rules. In Section 5, such problems are shown to be unitary, which is a nice property in particular when using anti-patterns for programming purposes. We finally report in the Section 6 on the implementation of this algorithm in Tom — a programming language that extends C and Java by offering algebraic data-types and pattern matching facilities [19,15] — and discuss how anti-patterns could be used to extend the expressiveness of this language.

Although we will make precise our main notations, we assume that the reader is familiar with the standard notions of algebraic rewrite systems, for example presented in [5,14].

[1] `http://www.ilog.com/products/jrules`

2 Terms and anti-terms

We briefly recall or introduce the notations for a few concepts that will be used along this paper.

A signature $\mathcal{F}$ is a set of function symbols, each one having a fixed arity. $\mathcal{T}(\mathcal{F},\mathcal{X})$ is the set of *terms* built from a given finite set $\mathcal{F}$ of function symbols and a denumerable set $\mathcal{X}$ of variables. A term t is said to be *linear* if no variable occurs more than once in t. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is empty, t is called a *ground term* and $\mathcal{T}(\mathcal{F})$ is the set of ground terms.

A *substitution* σ is an assignment from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F},\mathcal{X})$, denoted $\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$ when its domain $\mathsf{Dom}(\sigma)$ is finite. Its application, written $\sigma(t)$, is defined by $\sigma(x_i) = t_i$, $\sigma(f(t_1,\ldots,t_n)) = f(\sigma(t_1),\ldots,\sigma(t_n))$ for $f \in \mathcal{F}_n$, and $\sigma(y) = y$ if $y \notin \mathcal{D}om(\sigma)$. Given a term t, σ is called a *grounding substitution* when $\sigma(t) \in \mathcal{T}(\mathcal{F})$. The set of substitutions is denoted Σ. The set of grounding substitutions for a term t is denoted $\mathcal{GS}(t)$.

The ground semantics of a term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ is the set of all its ground instances: $[\![t]\!]_g = \{\sigma(t) \mid \sigma \in \mathcal{GS}(t)\}$. In particular, when $x \in \mathcal{X}$, we have $[\![x]\!]_g = \mathcal{T}(\mathcal{F})$.

2.1 Anti-terms

Definition 2.1 (Syntax of anti-terms). *Given $\mathcal{F}$ and $\mathcal{X}$, the syntax of an anti-term is defined as follows:*

$$\mathcal{AT} ::= x \;\mid\; \neg\mathcal{AT} \;\mid\; f(\mathcal{AT},\ldots,\mathcal{AT})$$

where $x \in \mathcal{X}$, $f \in \mathcal{F}$ and the arity is respected. The set of anti-terms is denoted $\mathcal{AT}(\mathcal{F},\mathcal{X})$ (resp. $\mathcal{AT}(\mathcal{F})$ for ground anti-terms). Any term is an anti-term, i.e. $\mathcal{T}(\mathcal{F},\mathcal{X}) \subseteq \mathcal{AT}(\mathcal{F},\mathcal{X})$.

For example, if x, y, z denote variables, a, b, c constants, f, g two function symbols of arity 2 and 1, the following expressions are anti-terms: $\neg x$, $\neg a$, $\neg f(\neg a, g(\neg x))$, $f(x,y)$, $f(\neg a, b)$, $f(x, \neg x)$.

Definition 2.2 (Free variables). *The free variables of an anti-term q are defined inductively by:*

1. $\mathcal{FV}ar(x) = \{x\}$,
2. $\mathcal{FV}ar(\neg q) = \emptyset$,
3. $\mathcal{FV}ar(f(q_1,\ldots,q_n)) = \cup_{i=1..n}\mathcal{FV}ar(q_i)$, *with the arity of f equal to n.*

Example 2.1. Assuming that a is a constant and f is binary, we have: $\mathcal{FV}ar(a) = \emptyset$, $\mathcal{FV}ar(\neg x) = \emptyset$, $\mathcal{FV}ar(f(x, \neg x)) = \{x\}$, $\mathcal{FV}ar(\neg f(x, \neg x)) = \emptyset$.

Definition 2.3 (Substitutions on anti-terms). *A substitution σ uniquely extends to an endomorphism σ' of $\mathcal{AT}(\mathcal{F},\mathcal{X})$: if x is a free variable, $\sigma'(x) = \sigma(x)$, otherwise $\sigma'(x) = x$. For $q, q_1, \ldots, q_n \in \mathcal{AT}(\mathcal{F},\mathcal{X})$, we have $\sigma'(f(q_1,\ldots,q_n)) = f(\sigma'(q_1),\ldots,\sigma'(q_n))$, and $\sigma'(\neg q) = \neg\sigma'(q)$.*

Example 2.2. Note that substitutions are active only on the free variables: $\sigma(f(x, \neg x)) = f(\sigma(x), \neg\sigma(x))$, $\sigma(f(x, \neg y)) = f(\sigma(x), \neg y)$.

The notion of grounding substitutions is also extended to anti-terms (e.g. t) as substitutions (e.g. σ) such that $\mathcal{FV}ar(\sigma(t)) = \emptyset$.

Intuitively, the semantics of the complement of a term represents the complement of its semantics in $\mathcal{T}(\mathcal{F})$. Therefore, the complement of a variable $\neg x$ denotes $\mathcal{T}(\mathcal{F})\backslash[\![x]\!]_g = \mathcal{T}(\mathcal{F})\backslash\mathcal{T}(\mathcal{F}) = \emptyset$. Similarly, $\neg f(x)$ denotes $\mathcal{T}(\mathcal{F})\backslash\{f(t) \mid t \in \mathcal{T}(\mathcal{F})\}$. In the following we extend this intuition to complements of complements, as well as complements which occur in subterms, and we formally define the semantics of an anti-term.

As usual, a *position* is a finite sequence of natural numbers. The subterm u of a term t at position ω is denoted $t_{|\omega}$, where ω describes the path from the root of t to the root of u. $t(\omega)$ denotes the root symbol of $t_{|\omega}$.

By $t[s]_\omega$ we express that the term t contains s as subterm at position ω. Positions are ordered in the classical way: $\omega_1 < \omega_2$ if ω_1 is the prefix of ω_2 [14].

The ground semantics extends to anti-terms:

Definition 2.4 (Ground semantics of anti-terms). *The ground semantics of any anti-term $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$ is defined recursively in the following way:*

$$[\![q[\neg q']_\omega]\!]_g = [\![q[z]_\omega]\!]_g \backslash [\![q[q']_\omega]\!]_g$$

where z is a fresh variable and for all $\omega' < \omega$, $q(\omega') \neq \neg$.

Example 2.3.

1. $[\![\neg a]\!]_g = [\![z]\!]_g \backslash [\![a]\!]_g = \mathcal{T}(\mathcal{F})\backslash\{a\}$,
2. $[\![\neg x]\!]_g = [\![z]\!]_g \backslash [\![x]\!]_g = \mathcal{T}(\mathcal{F})\backslash\mathcal{T}(\mathcal{F}) = \emptyset$, for any variable x,
3. $[\![\neg\neg x]\!]_g = [\![z]\!]_g \backslash [\![\neg x]\!]_g = [\![z]\!]_g \backslash ([\![z']\!]_g \backslash [\![x]\!]_g) = \mathcal{T}(\mathcal{F})\backslash(\mathcal{T}(\mathcal{F})\backslash\mathcal{T}(\mathcal{F})) = \mathcal{T}(\mathcal{F})$,
4. $[\![\neg g(x)]\!]_g = [\![z]\!]_g \backslash [\![g(x)]\!]_g = \mathcal{T}(\mathcal{F})\backslash\{g(\sigma(x)) \mid \sigma \in \mathcal{GS}(g(x))\}$,
5. $[\![g(\neg x)]\!]_g = [\![g(z)]\!]_g \backslash [\![g(x)]\!]_g = \emptyset$,
6. $[\![\neg g(\neg x)]\!]_g = [\![z]\!]_g \backslash [\![g(\neg x)]\!]_g = \mathcal{T}(\mathcal{F})\backslash\emptyset = \mathcal{T}(\mathcal{F})$,
7. we can also express that we are looking for something that is either not rooted by g, or it is $g(a)$:
$$\begin{aligned}[\![\neg g(\neg a)]\!]_g &= [\![z]\!]_g \backslash [\![g(\neg a)]\!]_g = [\![z]\!]_g \backslash ([\![g(z')]\!]_g \backslash [\![g(a)]\!]_g) \\ &= \mathcal{T}(\mathcal{F})\backslash([\![g(z')]\!]_g \backslash \{g(a)\}) \\ &= \mathcal{T}(\mathcal{F})\backslash(\{g(\sigma(z')) \mid \sigma \in \mathcal{GS}(g(z'))\}\backslash\{g(a)\}) \\ &= \mathcal{T}(\mathcal{F})\backslash\{g(z) \mid z \in \mathcal{T}(\mathcal{F}, \mathcal{X})\} \ \cup \ \{g(a)\},\end{aligned}$$
8. $[\![f(a, \neg b)]\!]_g = [\![f(a, z)]\!]_g \backslash [\![f(a, b)]\!]_g = \{f(a, \sigma(z)) \mid \sigma \in \mathcal{GS}(f(a, z))\}\backslash\{f(a, b)\}$,
9. $[\![\neg f(x, x)]\!]_g = [\![z]\!]_g \backslash [\![f(x, x)]\!]_g = \mathcal{T}(\mathcal{F})\backslash\{f(\sigma(x), \sigma(x)) \mid \sigma \in \mathcal{GS}(f(x, x))\}$
note the crucial use of non-linearity to denote any term except those rooted by f with identical subterms,
10. $$\begin{aligned}[\![f(x, \neg x)]\!]_g &= [\![f(x, z)]\!]_g \backslash [\![f(x, x)]\!]_g \\ &= \{f(\sigma(x), \sigma(z)) \mid \sigma \in \mathcal{GS}(f(x, z))\}\backslash\{f(\sigma(x), \sigma(x)) \mid \sigma \in \mathcal{GS}(f(x, x))\} \\ &= f(a, b), f(a, c), f(b, c), \ldots\end{aligned}$$

The second condition of Definition 2.4 is essential. It prevents from replacing a subterm by a fresh variable inside a complemented context (i.e. below a $\neg$). Otherwise, for $\neg g(\neg a)$ we would have had $[\![\neg g(\neg a)]\!]_g = [\![\neg g(z)]\!]_g \backslash [\![\neg g(a)]\!]_g = \emptyset$.

These simple examples show that anti-terms provide a compact and expressive representation for the sets of terms. A nice property can be easily derived from them:

Proposition 2.1. *For any $t \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$, we have $[\![\neg\neg t]\!]_g = [\![t]\!]_g$*

Proof. Using the Definition 2.4, $[\![\neg\neg t]\!]_g = [\![z]\!]_g \backslash [\![\neg t]\!]_g = [\![z]\!]_g \backslash ([\![z']\!]_g \backslash [\![t]\!]_g) = [\![t]\!]_g$. □

3 Matching anti-patterns

Before showing how anti-terms can be used for matching ground terms, we recall the standard definitions and results for the classical terms, as they are presented in [5,14] for example.

3.1 Pattern matching

Definition 3.1 (Matching).

1. *a* pattern *is a term,*
2. *a* matching equation *is a problem $p \prec\!\!\!\!\ll t$ with p a pattern and t a term,*
3. *a substitution σ is a* solution of the matching equation $p \prec\!\!\!\!\ll t$ *if $\sigma(p) = t$,*
4. *a* matching system S *is a conjunction of matching equations,*
5. *a substitution σ is a* solution of a matching system S *if it is solution of all the matching equations in S. The set of solutions of S is denoted by $\mathcal{S}ol(\mathsf{S})$,*
6. *we denote by Fail a matching system without solution.*

In this paper, without loss of generality, we only consider matching equations of the form $p \prec\!\!\!\!\ll t$ where t is a *ground term*. The solution of a matching system S, when it exists, is unique and is computed by a simple recursive algorithm [13]. This algorithm can be expressed by the set of *rewrite rules* Match, given below. The symbol $\wedge$ is assumed to be associative, commutative and idempotent, S is any conjunction of matching equations, p_i are patterns, and t_i are ground terms:

Decompose	$f(p_1, \ldots, p_n) \prec\!\!\!\!\ll f(t_1, \ldots, t_n)$	$\twoheadrightarrow$	$\bigwedge_{i=1,\ldots,n} p_i \prec\!\!\!\!\ll t_i$
SymbolClash	$f(p_1, \ldots, p_n) \prec\!\!\!\!\ll g(t_1, \ldots, t_m)$	$\twoheadrightarrow$	*Fail* if $f \neq g$
MergingClash	$x \prec\!\!\!\!\ll t_1 \wedge x \prec\!\!\!\!\ll t_2$	$\twoheadrightarrow$	*Fail* if $t_1 \neq t_2$
Delete	$p \prec\!\!\!\!\ll p$	$\twoheadrightarrow$	*True*
PropagateClash	$\mathsf{S} \wedge$ *Fail*	$\twoheadrightarrow$	*Fail*
PropagateSuccess	$\mathsf{S} \wedge$ *True*	$\twoheadrightarrow$	S

The soundness and the completeness of Match is expressed as follows:

Theorem 3.1 ([14]). *The normal form by the rules in* Match *of any matching problem $p \ll t$ such that $t \in \mathcal{T}(\mathcal{F})$, exists and is unique.*

1. *if it is of the form $\bigwedge_{i\in I} x_i \ll t_i$ with $I \neq \emptyset$, then the substitution $\sigma = \{x_i \mapsto t_i\}_{i\in I}$ is the unique match from p to t,*
2. *if it is True then p and t are identical, i.e. $p = t$,*
3. *if it is Fail, then there is no match from p to t.*

3.2 Anti-pattern matching

We now extend the classical notion of matching equation by allowing anti-terms on the left side. We will further call them anti-patterns.

When considering classical patterns, a matching equation $p \ll t$ has a solution when there exists a substitution σ such that $\sigma(p) = t$, that is when $t \in [\![p]\!]_g$. Indeed more precisely $\sigma \in \mathcal{GS}(p)$ is a solution if $\{t\} = [\![\sigma(p)]\!]_g$. This extends naturally to the anti-patterns.

Definition 3.2 (Solutions of anti-pattern matching). *For all $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$ and $t \in \mathcal{T}(\mathcal{F})$, the solutions of the anti-pattern matching problem $q \ll t$ are:*

$$Sol(q \ll t) = \{\sigma \mid t \in [\![\sigma(q)]\!]_g, \text{ with } \sigma \in \mathcal{GS}(q)\}$$

Remember that by Definition 2.3, the substitutions apply only on free variables. Also note that for $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, we have $[\![\sigma(p)]\!]_g = \{\sigma(p)\}$; this is not always true for the anti-patterns. Take for example $f(x, \neg b)$, and $\sigma = \{x \mapsto a\}$: the set $[\![\sigma(f(x, \neg b))]\!]_g = [\![f(a, \neg b))]\!]_g$ has more than one element, as we saw in Example 2.3. Here are some examples for the solutions of anti-pattern matching problems:

Example 3.1.

1. $Sol(f(a, \neg b) \ll f(a, a)) = \Sigma$,
2. $Sol(\neg g(x) \ll g(a)) = \{\sigma \mid g(a) \in \mathcal{T}(\mathcal{F}) \backslash \{g(\sigma(x)) \mid \sigma \in \mathcal{GS}(g(x))\}\} = \emptyset$,
3. $Sol(f(\neg a, x) \ll f(b, c)) = \{x \mapsto c\}$,
4. $Sol(f(x, \neg x) \ll f(a, b)) = \{x \mapsto a\}$,
5. $Sol(f(x, \neg g(x)) \ll f(a, g(b))) = \{x \mapsto a\}$,
6. $Sol(f(x, \neg g(x)) \ll f(a, g(a))) = \emptyset$.

4 Anti-pattern matching and equational problems

The relation between anti-pattern matching and equational problems is not trivial. For instance, the interpretation of $\neg q \ll t$ should not be $q \neq t$. Although this may be correct in the case of ground terms, like $\neg a \ll b$, it is not true in the general case. Take for example $\neg g(x) \ll g(a)$, which according to Definition 3.2 has no solution. But the solutions of $g(x) \neq g(a)$ are the solutions of $x \neq a$. In this section we provide a way of transforming any anti-pattern matching problem into a corresponding equational one that has the same set of solutions. We extend the notion of an equation between terms [11] to the notion of an equation containing anti-patterns:

Definition 4.1 (Solutions of equations with anti-patterns). *For any anti-pattern q and ground term t, σ is a solution of the equational problem $\exists w_1, \ldots, w_n, \forall y_1, \ldots, y_m : q = t$ if:*

1. *the domain of σ is $\mathcal{FV}ar(q) \backslash \{w_1, \ldots, w_n, y_1, \ldots, y_m\}$,*
2. *there exists a substitution ρ whose domain is $\{w_1, \ldots, w_n\} \backslash (\mathcal{FV}ar(q) \cup \{y_1, \ldots, y_m\})$ such that for all substitutions θ whose domain is $\{y_1, \ldots, y_m\} \backslash (\mathcal{FV}ar(q) \cup \{w_1, \ldots, w_n\})$ we have: $t \in [\![\theta\rho\sigma(q)]\!]_g$.*

We denote by $Sol(\exists w_1, \ldots, w_n, \forall y_1, \ldots, y_m : q = t)$ the set of all substitutions that are solutions of $\exists w_1, \ldots, w_n, \forall y_1, \ldots, y_m : q = t$. We have the following properties:

1. $Sol(q = t) = Sol(q \prec\!\!\prec t)$, since t is a ground term,
2. $Sol(\exists w_1, \ldots, w_n, \forall y_1, \ldots, y_m : not(q = t)) = \{\sigma \mid t \notin [\![\theta\rho\sigma(q)]\!]_g\}$, with the same conditions on θ, ρ, σ as in Definition 4.1, and *not* being the classical logic negation. One may notice that the substitutions ρ and σ do not have the variables $\{y_1, \ldots, y_m\}$ in their domains, and therefore we can safely eliminate θ in $Sol(\exists w_1, \ldots, w_n, \forall y_1, \ldots, y_m : not(q = t)) = \{\sigma \mid t \notin [\![\rho\sigma(q)]\!]_g\}$, because the ground semantics will instantiate anyway $\{y_1, \ldots, y_m\}$ with all their possible values.

Given an anti-pattern q and a ground term t, we consider the following rewrite system AP-Elim. This transforms an anti-pattern matching problem into an equational one:

$$
\begin{array}{lll}
\textsf{ElimMatch} & q \prec\!\!\prec t & \twoheadrightarrow\!\!\!\!\mapsto \; q = t \\
\textsf{ElimAnti} & q[\neg q']_\omega = t & \twoheadrightarrow\!\!\!\!\mapsto \; \exists z \; q[z]_\omega = t \; \wedge \; \forall x \in \mathcal{FV}ar(q') \; not(q[q']_\omega = t) \\
 & & \quad \text{if } \forall \; \omega' < \omega, \; q(\omega') \neq \neg \text{ and } z \text{ a fresh variable}
\end{array}
$$

Clearly, these rules are terminating and the normal form does not contain anymore the $\neg$ symbol. If we apply these rules on the example we provided earlier, $\neg g(x) \prec\!\!\prec g(a)$, we obtain $\exists z \; z = g(a) \; \wedge \; \forall x \; not(g(x) = g(a))$ which is equivalent with $\forall x \; g(x) \neq g(a)$, that has no solution. Thus, for this example these transformations are valid. As shown below they are also valid in the general case:

Proposition 4.1. *The rules are sound and preserving: they do not introduce unexpected solutions, and no solution is lost in the application of the rules.*

Proof. By Definition 4.1, this is clear for the rule ElimMatch. For ElimAnti, we consider ω a position such that $q[\neg q']_\omega$ and $\forall \; \omega' < \omega, \; q(\omega') \neq \neg$.

Considering as usual that $Sol(A \wedge B) = Sol(A) \cap Sol(B)$ we have the following result for the right hand side of the rule:

$$
\begin{array}{l}
Sol(\exists z \; q[z]_\omega = t \; \wedge \; \forall x \in \mathcal{FV}ar(q') \; not(q[q']_\omega = t)) \\
= Sol(\exists z \; q[z]_\omega = t \;) \cap \; Sol(\forall x \in \mathcal{FV}ar(q') \; not(q[q']_\omega = t))
\end{array}
$$

From Definition 4.1, $Sol(\exists z\ q[z]_\omega = t) = \{\sigma \mid \exists \rho \text{ such that } \mathsf{Dom}(\rho) = \{z\}, t \in [\![\rho\sigma(q[z]_\omega)]\!]_g,$ and $\mathsf{Dom}(\sigma) = \mathcal{FV}ar(q[z])\backslash\{z\}\}$.

To have $t \in [\![\rho\sigma(q[z]_\omega)]\!]_g$ the only possible value for $\rho(z)$ is $t_{|\omega}$. So we can further rewrite the above solutions in:

$$\{\sigma \mid t \in [\![\sigma(q[t_{|\omega}]_\omega)]\!]_g,\ with\ \mathsf{Dom}(\sigma) = \mathcal{FV}ar(q[z])\backslash\{z\}\} \tag{1}$$

Applying also the Definition 4.1, $Sol(\forall x \in \mathcal{FV}ar(q')\ not(q[q']_\omega = t))$ is equal to:

$$\{\sigma \mid t \notin [\![\sigma(q[q']_\omega)]\!]_g\ with\ \mathsf{Dom}(\sigma) = \mathcal{FV}ar(q[q']) \setminus \mathcal{FV}ar(q')\} \tag{2}$$

On the other hand, for the left part of the rule ElimAnti, by Definition 4.1 we have:

$$\begin{aligned} Sol\ (q[\neg q']_\omega = t) &= \{\sigma \mid t \in [\![\sigma(q[\neg q']_\omega)]\!]_g,\ with\ \mathsf{Dom}(\sigma) = \mathcal{FV}ar(q[\neg q'])\} \\ &= \{\sigma \mid t \in ([\![\sigma(q[z]_\omega)]\!]_g \backslash [\![\sigma(q[q']_\omega)]\!]_g),\ with \ldots\},\ \text{since } \forall \omega' < \omega, q(\omega') \neq \neg \\ &= \{\sigma \mid t \in [\![\sigma(q[z]_\omega)]\!]_g\ and\ t \notin [\![\sigma(q[q']_\omega)]\!]_g,\ with\ \mathsf{Dom}(\sigma) = \mathcal{FV}ar(q[\neg q'])\} \\ &= \{\sigma \mid t \in [\![\sigma(q[z]_\omega)]\!]_g,\ with \ldots\} \cap\ \{\sigma \mid t \notin [\![\sigma(q[q']_\omega)]\!]_g\ with \ldots\} \end{aligned} \tag{3}$$

Now it remains to check the equivalence of (3) with the intersection of (1) and (2). First of all, $\mathcal{FV}ar(q[z])\backslash\{z\} = \mathcal{FV}ar(q[q']) \setminus \mathcal{FV}ar(q') = \mathcal{FV}ar(q[\neg q'])$ which means that we have the same domain for σ in (3), (1), and (2). Therefore, we have to prove: $\{\sigma \mid t \in [\![\sigma(q[z]_\omega)]\!]_g\} = \{\sigma \mid t \in [\![\sigma(q[t_{|\omega}]_\omega)]\!]_g\}$.

But σ does not instantiate z, and for the inclusion $t \in [\![\sigma(q[z]_\omega)]\!]_g$ to be true, the only possible value of z is $t_{|\omega}$. As we considered an arbitrary $\neg$, we can conclude that the rule is sound and preserving, wherever it is applied on a term. □

Using the rewrite system AP-Elim, we can eliminate all $\neg$ symbols from any anti-pattern matching problem. The normal forms have the following structure: $\exists z\ q = t\ \wedge\ \forall x\ not(\exists z'\ q' = t\ \wedge\ \forall x'\ not(\ldots))$.

We consider a set of boolean simplification rules, called DeMorgan, that is applied on these normal forms: $not(\exists z\ P) \twoheadrightarrow \forall z\ not(P)$, $not(\forall z\ P) \twoheadrightarrow \exists z\ not(P)$, $not(a \wedge b) \twoheadrightarrow not(a) \vee not(b)$, $not(a \vee b) \twoheadrightarrow not(a) \wedge not(b)$, $not(not(a)) \twoheadrightarrow a$, $not(a = b) \twoheadrightarrow a \neq b$, $not(a \neq b) \twoheadrightarrow a = b$. The resulting expression no longer contains any *not*, and thus is a classical equational problem. We call it an *anti-pattern disunification problem*.

5 Solving anti-pattern matching via disunification

As presented previously, an anti-pattern matching problem can be translated into an equivalent equational problem. A natural way to solve this type of problem is to use a disunification algorithm such as described in [11]. Due to lack of space, we cannot present disunification in detail. Instead we give in Figure 1 the set of rules we consider. The interested reader can refer to [11] for a detailed presentation of disunification.

$\mathsf{Universality}_1$	$\forall z:\ z = t \wedge \mathsf{S}$	$\mapsto\!\!\!\!\mapsto$	$\bot$
$\mathsf{Universality}_2$	$\forall z:\ z \neq t \wedge \mathsf{S}$	$\mapsto\!\!\!\!\mapsto$	$\bot$
$\mathsf{Universality}_3$	$\forall z:\ \mathsf{S}$	$\mapsto\!\!\!\!\mapsto$	S if $z \notin \mathcal{V}ar(\mathsf{S})$
$\mathsf{Universality}_4$	$\forall z:\ \mathsf{S} \wedge (z \neq t \vee \mathsf{S}')$	$\mapsto\!\!\!\!\mapsto$	$\forall z:\ \mathsf{S} \wedge \mathsf{S}'(z \leftarrow t)$
$\mathsf{Universality}_5$	$\forall z:\ \mathsf{S} \wedge (z = t \vee \mathsf{S}')$	$\mapsto\!\!\!\!\mapsto$	$\forall z:\ \mathsf{S} \wedge \mathsf{S}'$ if $z \notin \mathcal{V}ar(\mathsf{S}')$
$\mathsf{Replacement}$	$z = t \wedge \mathsf{S}$	$\mapsto\!\!\!\!\mapsto$	$z = t \wedge \mathsf{S}(z \leftarrow t)$
$\mathsf{Elimination}_1$	$a = a$	$\mapsto\!\!\!\!\mapsto$	$\top$
$\mathsf{Elimination}_2$	$a \neq a$	$\mapsto\!\!\!\!\mapsto$	$\bot$
$\mathsf{PropagateClash}_1$	$\mathsf{S} \wedge \bot$	$\mapsto\!\!\!\!\mapsto$	$\bot$
$\mathsf{PropagateClash}_2$	$\mathsf{S} \vee \bot$	$\mapsto\!\!\!\!\mapsto$	S
$\mathsf{PropagateSuccess}_1$	$\mathsf{S} \wedge \top$	$\mapsto\!\!\!\!\mapsto$	S
$\mathsf{PropagateSuccess}_2$	$\mathsf{S} \vee \top$	$\mapsto\!\!\!\!\mapsto$	$\top$
Clean_1	$a \wedge a$	$\mapsto\!\!\!\!\mapsto$	a
Clean_2	$a \vee a$	$\mapsto\!\!\!\!\mapsto$	a
Clash_1	$f(p_1 \dots p_n) = g(t_1 \dots t_n)$	$\mapsto\!\!\!\!\mapsto$	$\bot$ if $f \not\equiv g$
Clash_2	$f(p_1 \dots p_n) \neq g(t_1 \dots t_n)$	$\mapsto\!\!\!\!\mapsto$	$\top$ if $f \not\equiv g$
$\mathsf{Decompose}_1$	$f(p_1 \dots p_n) = f(t_1 \dots t_n)$	$\mapsto\!\!\!\!\mapsto$	$\bigwedge_{i=1,\dots,n} p_i = t_i$
$\mathsf{Decompose}_2$	$f(P_1 \dots P_n) \neq f(t_1 \dots t_n)$	$\mapsto\!\!\!\!\mapsto$	$\bigvee_{i=1,\dots,n} p_i \neq t_i$
$\mathsf{Merging}_1$	$z = t \wedge z = u$	$\mapsto\!\!\!\!\mapsto$	$z = t \wedge t = u$
$\mathsf{Merging}_2$	$z \neq t \vee z \neq u$	$\mapsto\!\!\!\!\mapsto$	$z \neq t \vee t \neq u$
$\mathsf{Merging}_3$	$z = t \wedge z \neq u$	$\mapsto\!\!\!\!\mapsto$	$z = t \wedge t \neq u$
$\mathsf{Merging}_4$	$z = t \vee z \neq u$	$\mapsto\!\!\!\!\mapsto$	$t = u \vee z \neq u$
Removed rules:	$\mathsf{OccurCheck}, \mathsf{Explosion}, \mathsf{Elimination\ of\ disjunctions}$		
New rules:			
Exists_1	$\exists z:\ \mathsf{S}$	$\mapsto\!\!\!\!\mapsto$	S if $z \notin \mathcal{V}ar(\mathsf{S})$
Exists_2	$\exists z:\ \mathsf{S} \wedge (z \neq t \vee \mathsf{S}')$	$\mapsto\!\!\!\!\mapsto$	S if $z \notin \mathcal{V}ar(\mathsf{S})$
Exists_3	$\exists z:\ \mathsf{S} \wedge (z = t \vee \mathsf{S}')$	$\mapsto\!\!\!\!\mapsto$	S if $z \notin \mathcal{V}ar(\mathsf{S})$

Fig. 1. Simplified presentation of the disunification rules: AP-Match

5.1 Disunification rules

[11] presents a set of disunification rules that is proved to be sound and preserving. Moreover, irreducible problems for these rules are definitions with constraints, i.e. either $\top$, $\bot$ or a conjunction of equalities and disequalities. In Figure 1 we present this set of rules, but tailored for anti-pattern matching problems. It is still sound and preserving, but also ensures (thanks to Theorem 5.1) that for each problem a normal form exists and is unique. We will further call it AP-Match.

From the classical presentation of disunification rules, three rules have been removed. They were no longer necessary in the restricted case of the anti-patterns, as their application conditions are never fulfilled. Three new rules that are proved to be sound and preserving [9] have been added. They ensure the elimination of all variables that are existentially quantified. The justification is simple, and consists in showing that any problem containing an occurrence of

an existentially quantified variable is reducible: if there is such a variable, one of the three introduced rules is tried. The condition $z \notin \mathcal{V}ar(\mathsf{S})$ may prevent from applying a rule. In that case, we have $z \in \mathcal{V}ar(\mathsf{S})$ and therefore one of the following rules can be applied: Replacement (or Merging), Decompose (or Clash) — if the variable z is inside a term.

In [11] there is a clear separation between the elimination of parameters and the rules that reach definitions with constraints. But, as affirmed both in [11] and [9], such a strict control is only for presentation purposes. In our algorithm, we use a single step approach.

5.2 Solved forms

In the following we show that an *anti-pattern disunification problem* (resulting from the application of AP-Elim, followed by DeMorgan can be simplified by the rewrite system AP-Match, given in Figure 1, such that it does not contain any disjunction or disequality.

Example 5.1. If we consider $f(x, \neg y) \prec\!\!\!\prec f(a, b)$, the corresponding anti-pattern disunification problem is computed in the following way:

$$\begin{aligned} f(x, \neg y) \prec\!\!\!\prec f(a, b) &\mapsto\!\!\!\!\mapsto f(x, \neg y) = f(a, b) \\ &\mapsto\!\!\!\!\mapsto \exists z\ f(x, z) = f(a, b) \wedge \forall y\ not(f(x, y) = f(a, b)) \\ &\mapsto\!\!\!\!\mapsto \exists z\ f(x, z) = f(a, b) \wedge \forall y\ f(x, y) \neq f(a, b) \end{aligned}$$

Proposition 5.1. *Given an* anti-pattern disunification problem*, the normal form wrt. the rewrite system* AP-Match *does not contain disjunctions or disequalities.*

Proof. We consider an anti-pattern $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$, and an arbitrary application of ElimAnti:

$$q[\neg q']_\omega = t \ \mapsto\!\!\!\!\mapsto\ \exists z\ q[z]_\omega = t\ \wedge\ \forall x \in \mathcal{FV}ar(q')\ not(q[q']_\omega = t)$$

If a disequality or a disjunction is produced, it comes from the $not(q[q']_\omega = t)$. We now consider the variables that occur in this expression. Each of them belongs to one of the following classes:

1. the free variables of q',
2. the free variables of $q[q']_\omega$ — excepting the free variables of q',
3. the variables of $q[q']_\omega$ that are not free.

In the following we show that the normal form cannot contain such a variable. Therefore, the normalization of $\forall x \in \mathcal{FV}ar(q'), not(q[q']_\omega = t)$ leads to either $\top$ or $\bot$:

1. these are universally quantified variables, and they will be eliminated by Universality rules,

2. let us consider $y \in \mathcal{FV}ar(q[q']_\omega) \backslash \mathcal{FV}ar(q')$, and let us suppose that the reduction of $not(q[q']_\omega = t)$ generates the disequality $y \neq t_{|\omega_1}$, then the reduction of the first part $\exists z\ q[z]_\omega = t$ will generate $y = t_{|\omega_2}$, with $\omega_2 = \omega_1$ because t and the skeleton of q are the same in both parts. By applying the Replacement rule, all the occurrences of $y \neq t_{|\omega_1}$ are transformed in $t_{|\omega_1} \neq t_{|\omega_1}$ and later eliminated,
3. any variable that is not free (i.e. is under a $\neg$) will be universally quantified by a further application of the rule ElimAnti, therefore later eliminated by $\mathsf{Universality}_1$ or $\mathsf{Universality}_2$. □

Theorem 5.1. *Given an* anti-pattern disunification problem*, its normal form wrt. the rewrite system* AP-Match *exists and is unique.*

1. *when it is of the form* $\bigwedge_{i\in I} x_i = t_i$ *with* $I \neq \emptyset$ *and* $x_i \neq x_j$ *for all* $i \neq j$*, the substitution* $\sigma = \{x_i \mapsto t_i\}_{i\in I}$ *is the solution of the matching problem,*
2. *when it is* $\top$*, any substitution* σ *is a solution of the matching problem,*
3. *when it is* $\bot$*, the matching problem has no solution.*

Proof. By applying Proposition 5.1. □

5.3 Simple examples

Let us show on a few examples how the rules behave. First with one complement:
$f(a, \neg b) \prec\!\!\prec f(a, a)$
$\mapsto\!\!\!\!\rightarrow\ f(a, \neg b) = f(a, a) \mapsto\!\!\!\!\rightarrow \exists z f(a, z) = f(a, a) \wedge not(f(a, b) = f(a, a))$
$\mapsto\!\!\!\!\rightarrow \exists z f(a, z) = f(a, a) \wedge f(a, b) \neq f(a, a)$
$\mapsto\!\!\!\!\rightarrow \exists z (a = a \wedge z = a) \wedge (a \neq a \vee b \neq a) \mapsto\!\!\!\!\rightarrow \exists z (z = a) \wedge (\bot \vee \top)$
$\mapsto\!\!\!\!\rightarrow \top \wedge \top \mapsto\!\!\!\!\rightarrow \top.$
Of course complements can be nested as illustrated below:
$\neg f(a, \neg b) \prec\!\!\prec f(a, b)$
$\mapsto\!\!\!\!\rightarrow \neg f(a, \neg b) = f(a, b) \mapsto\!\!\!\!\rightarrow \exists z\ z = f(a, b) \wedge not(f(a, \neg b) = f(a, b))$
$\mapsto\!\!\!\!\rightarrow \exists z\ z = f(a, b) \wedge not(\ \exists z' f(a, z') = f(a, b) \wedge not(f(a, b) = f(a, b)))$
$\mapsto\!\!\!\!\rightarrow \exists z\ z = f(a, b) \wedge (\forall z' f(a, z') = f(a, b) \vee f(a, b) = f(a, b))$
$\mapsto\!\!\!\!\rightarrow \top \wedge (\forall z' (a = a \wedge z' = b) \vee (a = a \wedge b = b))$
$\mapsto\!\!\!\!\rightarrow \forall z' (z' = b) \vee \top \mapsto\!\!\!\!\rightarrow \top.$
We can also consider anti-pattern problems with variables, such as $f(\neg a, x) \prec\!\!\prec f(b, c)$, whose solution is $\{x \mapsto c\}$. The pattern can be non-linear: $f(x, \neg x) \prec\!\!\prec f(a, b)$, leading to $\{x \mapsto a\}$. Nested negation and non-linearity can be combined:
$\neg f(x, \neg g(x)) \prec\!\!\prec f(a, g(b))$
$\mapsto\!\!\!\!\rightarrow \neg f(x, \neg g(x)) = f(a, g(b))$
$\mapsto\!\!\!\!\rightarrow \exists z\ z = f(a, g(b)) \wedge \forall x\ not(f(x, \neg g(x)) = f(a, g(b)))$
$\mapsto\!\!\!\!\rightarrow \exists z\ z = f(a, g(b)) \wedge \forall x\ not(\exists z'\ f(x, z') = f(a, g(b))$
$\qquad\qquad\qquad\qquad \wedge \forall x\ not(f(x, g(x)) = f(a, g(b))))$
$\mapsto\!\!\!\!\rightarrow \exists z\ z = f(a, g(b)) \wedge \forall x (\forall z'\ f(x, z') = f(a, g(b)) \vee \exists x\ f(x, g(x)) = f(a, g(b)))$
$\mapsto\!\!\!\!\rightarrow \top \wedge \forall x\ (\forall z'\ (x = a \wedge z' = g(b)) \vee \exists x\ (x = a \wedge g(x) = g(b)))$
$\mapsto\!\!\!\!\rightarrow \forall x\ (x = a \wedge \forall z'\ (z' = g(b)) \vee \exists x\ (x = a \wedge x = b))$
$\mapsto\!\!\!\!\rightarrow \forall x\ (x = a \wedge \bot \vee \exists x\ (x = a \wedge a = b))$
$\mapsto\!\!\!\!\rightarrow \forall x\ (\bot \vee \exists x\ (x = a \wedge \bot)) \mapsto\!\!\!\!\rightarrow \forall x\ (\bot \vee \bot) \mapsto\!\!\!\!\rightarrow \bot.$

5.4 Summing up the relations with disunification

When comparing anti-pattern problems with general disunification ones, there are many similarities, but some important differences also. In the anti-pattern case, a solved form does not contain any quantifier whereas disunification allows existential ones. Another important difference is the unitary property (Theorem 5.1) which is obviously not true for disunification: $x \neq a$ has many solutions in general. Disunification contains rules (called *globally preserving*) that return an equational problem whose solutions are a subset of the given problem. The Explosion and the Elimination of disjunctions rules are such examples. In our case, the complexity is dramatically reduced since these rules are unnecessary.

6 Implementation

We do not have enough space to present the implementation in detail but the reader should know that the presented anti-pattern matching algorithm has been fully implemented and integrated in Tom[2]. With the purpose of also supporting anti-patterns, we enriched the syntax of the Tom patterns to allow the use the operator '!' (representing '¬'). Therefore, constructs as the following one are now valid in this language:

```
%match(s) {
  f(a(),g(b()))  -> { /* action 1: executed when f(a,g(b))<<s  */ }
  f(!a(),g(b())) -> { /* action 2: when f(x,g(b))<<s with x!=a */ }
  !f(x,!g(x))    -> { /* action 3: when not f(x,y)<<s or ...   */ }
  !f(x,g(y))     -> { /* action 4 */ }
}
```

Similarly to `switch/case`, an action part is executed when its corresponding pattern matches the subject `s`. Note that non-linear patterns are allowed.

Without the use of anti-patterns, one would be forced to verify additional conditions in the action part. For example, the previous `%match` should have been written:

```
%match(s) {
  f(a(),g(b()))  -> { /* action 1 */ }
  f(x,g(b)) -> { if(x != a) { /* action 2 */ } }
  y -> { if(symb(y) != f) { /* action 3 */ }
         else { %match(y) { f(x,g(x)) -> { /* action 3 */ } } } }
  z -> {
    if(symb(z) != f) { /* action 4 */ }
    else { %match(z) {
             f(x,g(y)) -> { break; /* do not perform action 4 */ }
             _ -> { /* action 4 */ } } } }
}
```

[2] `http://tom.loria.fr`

This example clearly shows that anti-pattern semantics cannot be easily obtained in a standard setting. Note also that method extraction would be necessary to avoid duplicating actions. This would make the code even more complex.

7 Related work

There has been a huge amount of work that can be related in a way or another with the content of this paper. In spite of this, the anti-patterns are quite a novelty for pattern matching languages. It is important to stress that we introduced the anti-patterns with the purpose of having a compact and permissive representation to match *ground terms*: the use of nested negations replaces the use of conjunctions and/or disjunctions and there is no restriction to linear terms for example. It is also a useful representation which is both intuitive and easy to compile in an efficient way. In the context of Tom, general algorithms such as disunification [10,11,9] could have been used. But since pattern-matching is the main execution mechanism, we were interested in a specialized approach that is both simpler and more efficient.

Lassez [16] presented a way of expressing exclusion by the means of counter-examples: typically, the expression $f(x,y)/\{f(a,u) \vee f(u,a)\}$ represents all the ground instances of $f(x,y)$, different from $f(a,u)$ and $f(u,a)$. Even though this is a useful and close approach, it is more restrictive than the anti-patterns. Consider for example the anti-pattern $\neg f(a,\neg b)$, that cannot be represented by terms with counter-examples, unless we allow the counter-examples to also have counter-examples, i.e. $z/\{f(a,y/\{b\})\}$ — an issue not addressed in [16]. Moreover, the application domain of terms with counter-examples was rather machine learning than efficient term rewriting. This may explain why they restricted to linear terms and studied if these types of expressions have an equivalent representation using disjunctions. Actually, complementing non-linear terms was not very much addressed (except for disunification) and standard algorithms that computes complements are incorrect for non-linear terms, as mentioned in [18]. Complementing higher order patterns is also considered only in the linear case.

Although the syntax of set constraints [2,20,1,7] allows the use of complement without any restriction of linearity or level of complement, we are not aware of any good semantics for the general case. Moreover, despite the fact that theoretically it is possible to have a constraint of the form $f(a,b) \subseteq \neg f(a,\neg b)$, existing implementations do not allow the complement in its fully generality. For example the CLP(Set) language in B-Prolog[3] allows the use of the symbol '\' as a unary operator representing the complement. However, it is only defined for variables, and not for constants. Another example is CLP(SC) [12], where we are restricted to use only predicates of arity 0 and 1, which obviously cannot have the same expressiveness as anti-patterns. Besides that, it does not provide variable assignments. Constraints over features trees [4,3,6] include the *exclusion constraint* which is a formula of the form $\neg\exists y(xfy)$, which says that the feature f

[3] `http://www.probp.com/`

is undefined for x, i.e. there is no edge that starts from x labeled with f. A more complex semantics of nested negations is not provided, for example to express that *there is no 'a' in relation with x, unless x is in relation with 'b'.*

CDuce[4] allows for the use of complement when declaring types but it restricts it to be used on types alone, and do not deal with variables complements.

The constrained terms, as defined in [9], can be used to obtain the semantics of some anti-patterns. They may have constraints — conjunction of disequalities — attached to their variables. Considering for example $f(a, \neg b)$, this is semantically equivalent to $f(a, z)$, *constrained by* $z \neq b$. But for a more complex expression, like $f(a, \neg g(b, \neg c))$, this approach is not expressive enough because the use of disjunctions in the constraints is not allowed.

8 Conclusion and future work

In this paper we have defined the notion of *anti-patterns* along with their semantics. We have shown how anti-pattern matching problems can be transformed in specific disunification problems. Therefore, most of the properties (confluence, termination) that hold for the disunification rules are still true for the anti-pattern matching ones. Moreover, we proved that anti-pattern matching is unitary, that the rules are sound and fully preserving, and that the computed solved forms do not contain any disequality — properties that are not true for general disunification problems. Finally, the anti-pattern matching algorithm has been implemented and is available in the Tom system.

We are currently working on two questions. The first one is about the precise complexity of the anti-pattern matching problem. For instance, the satisfiability in $\mathcal{T}(\mathcal{F})$ of equational problems is known to be NP-complete. However, solving anti-pattern matching being a more restricted disunification problem, we conjecture that solving an anti-pattern matching problem is polynomial.

The second one concerns the study of anti-pattern matching in presence of associative operators. This is quite appealing because of the nice expressiveness that such a feature will provide. For instance in Tom the pattern $(*, !a, *)$ would denote a list which contains at least one element different from a, whereas $!(*, a, *)$ would denote a list which does not contain any a. This will be more generally useful for theories like associativity and commutativity and anti-pattern matching should therefore be investigated for appropriate equational theories.

Acknowledgments: We sincerely thank Luigi Liquori for stimulating discussions and suggestions, Emilie Balland for her comments on the preliminary version of this paper and the anonymous referees for their valuable remarks and suggestions.

References

1. A. Aiken, D. Kozen, and E. Wimmers. Decidability of systems of set constraints with negative constraints. *Information and Computation*, 122(1):30–44, 1995.

[4] `http://www.cduce.org/`

2. A. Aiken and E. L. Wimmers. Solving systems of set constraints (extended abstract). In *LICS*, pages 329–340. IEEE Computer Society, 1992.
3. H. Ait-Kaci, A. Podelski, and G. Smolka. A feature constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1–2):263–283, 1994.
4. F. Baader, H.-J. Bürckert, B. Nebel, W. Nutt, and G. Smolka. On the expressivity of feature logics with negation, functional uncertainty, and sort equations. *Journal of Logic, Language and Information*, 2:1–18, 1993.
5. F. Baader and T. Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.
6. R. Backofen and G. Smolka. A complete and recursive feature theory. *Theoretical Computer Science*, 146(1–2):243–268, July 1995.
7. W. Charatonik and L. Pacholski. Negative set constraints with equality. In *LICS*, pages 128–136. IEEE Computer Society, 1994.
8. K. L. Clark. *Logic and databases*, chapter Negation as Failure, pages 293–322. Plenum Press, New York, 1978.
9. H. Comon. *Unification et disunification. Théories et applications.* Thèse de Doctorat d'Université, Institut Polytechnique de Grenoble (France), 1988.
10. H. Comon. Disunification: a survey. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 9, pages 322–359. The MIT press, Cambridge (MA, USA), 1991.
11. H. Comon and P. Lescanne. Equational problems and disunification. In C. Kirchner, editor, *Unification*, pages 297–352. Academic Press inc., London, 1990.
12. J. S. Foster. CLP(SC): Implementation and efficiency considerations. In *Proceedings Workshop on Set Constraints, held in Conjunction with CP'96, Boston, Massachusetts*, 1996.
13. G. Huet. *Résolution d'equations dans les langages d'ordre 1, 2, ..., ω.* Thèse de Doctorat d'Etat, Université de Paris 7 (France), 1976.
14. C. Kirchner and H. Kirchner. Rewriting, solving, proving. A preliminary version of a book available at `http://www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.
15. C. Kirchner, P.-E. Moreau, and A. Reilles. Formal validation of pattern matching code. In P. Barahona and A. Felty, editors, *Proceedings of the 7th ACM SIGPLAN PPDP*, pages 187–197. ACM, July 2005.
16. J.-L. Lassez and K. Marriott. Explicit representation of terms defined by counter examples. *Journal of Automated Reasoning*, 3(3):301–317, 1987.
17. L. Liquori. iRho: the software [system description]. *DCM: International Workshop on Development in Computational Models. Electr. Notes Theor. Comput. Sci.*, 135(3):85–94, 2006.
18. A. Momigliano. Elimination of negation in a logical framework. In *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic*, volume 1862 of *LNCS*, pages 411–426, London, UK, 2000. Springer Verlag.
19. P.-E. Moreau, C. Ringeissen, and M. Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.
20. M. Müller, J. Niehren, and A. Podelski. Inclusion constraints over non-empty sets of trees. In M. Dauchet, editor, *Theory and Practice of Software Development, International Joint Conference CAAP/FASE/TOOLS*, volume 1214 of *LNCS*, pages 217–231. Springer Verlag, Apr. 1997.